

confr – A Configuration System for Machine Learning Projects

Mattias Arro¹

¹www.uxo.ai

Abstract

Finding a performant machine learning model usually requires exploring different combinations of model hyperparameters, preprocessing steps, data generation and train logic. To facilitate a clear analysis of the factors that determine accuracy, it is useful to make the data processing and train pipeline highly configurable such that a combination of a code version and configuration file uniquely determines the behaviour of the system. A poor configuration system can lead to repetitive code that is hard to maintain, understand, and brittle due to insufficient configuration validation logic. This paper outlines the design and usage of `confr`, a concise and flexible configuration system geared towards Python-based machine learning projects. It combines some of the capabilities of commonly used systems (such as `gin-config`, `OmegaConf`, and `Hydra`) into a library which aims to reduce repetitive code and maintenance overhead. It can be used both as part of a notebook-based and script-based workloads, and can be used for ensuring that there is no accidental difference between inference-time and train-time behaviour.

Keywords

machine learning, configuration, experiment management, reproducibility

1. Introduction

The goal of machine learning (ML) practitioners is to find a "good model". This is broadly determined by three factors: (training and validation) data, code (preprocessing, model implementation, train loop), and hyperparameters (configuration). By hyperparameters we mean any non-learnable parameter/configuration that influences how the code gets executed, which may be in the data processing, model intialisation or inference, or train loop. When running experiments, exact and immutable versions of the three should always be stored, so that we can (1) analyse the factors that influence accuracy and (2) reproduce the results of an earlier experiment.

Early in developing a ML system, code tends to hard-code most choices for data processing, model implementation and hyperparameters. Experimentation in this setting would require changes to the code, which means that to compare two experiments one needs to find the differences of code used in each train run, or rely on the experimenter's description of the hypothesis that was tested. Given that most ML experiments are done in notebooks where comparing code with version control is difficult, comparing large numbers of such experiments is not feasible.

A better approach is to make the code highly configurable, so that alternative behaviours can be achieved by using different hyperparameter values. Now triggering

many train runs (on the same code and data version) creates a dataset of hyperparameter sets and corresponding evaluation metrics. The relationship between hyperparameters and metrics can be analysed for insights that inform future hyperparameter choices or code changes. A highly configurable train pipeline also lends itself for automatic model tuning approaches such as brute-force grid/random search, or methods like Bayesian optimisation that use ML to find values for hyperparameters which maximise validation accuracy.

There are many ways to make a system configurable, such as creating an ad hoc solution from scratch or using a 3rd party `config`¹ system. The following is a list of qualities we would expect from a config system, which we will later use to evaluate our proposed system `confr` against alternatives.

1. **Minimise boilerplate code.** There should not be much repetitive code to have a highly configurable system.
2. **Minimise repetitive config.** There should be ways to reuse, rather than repeat, individual config values.
3. **Composability of config objects.** It should be possible to reuse and compose different configurations, which is crucial for large systems.
4. **Low maintenance overhead.** A config system should reduce (rather than add to) the difficulty of refactoring and developing the code base.
5. **Low cognitive load.** A config system should make it easy to understand which variables are

ITAT'22: Information technologies – Applications and Theory, September 23–27, 2022, Zuberac, Slovakia

 mattias.arro@gmail.com (M. Arro)

 © 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License

Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

¹We use the words "config" and "configuration" interchangeably, as is common in industry.

configurable and where that configuration comes from. It is highly related (but not fully determined) by the following three qualities.

6. **Clearly identifiable configurable arguments.** It should be clear just by looking at a function if its argument is a configurable hyperparameter. This improves readability and decreases the chance of accidentally forgetting to provide a config value.
7. **Consistent mapping between config keys and variables / arguments.** The system should encourage a one-to-one mapping between keys² in the config file and configurable arguments. Such mapping alleviates the cognitive load in translating differences between config keys and variable names, and makes it easy to search for all places where a config key is used.
8. **Global config values.** In most cases we expect the value of a config key to be the same across the code base, and a config system should encourage this. For example, if a preprocessing function assumes `max_img_dimensions = (128, 128)`, then so should the function which builds the neural network whose input tensor would have the shape `(128, 128, 3)` - otherwise the model's input dimensions would be incompatible with the images created by the preprocessor.
9. **Centralised, multi-key validation of config.** Validation of config values should be centralised in a single place, rather than scattered around the project in an ad hoc manner. This (1) ensures we fail quickly with a helpful message when reading a faulty config (rather than waiting for the relevant code path to be reached, which may happen at a much later state), and (2) allows defining validation logic that sets constraints on several config keys simultaneously.
10. **Usable in a notebook as well as CLI.** The system should be easy to use in an exploratory notebook-based setting, where one might want to dynamically (re-)define and access config values, and to ultimately write the current active configuration to a file. It should also work in a script-oriented CLI³ setting, where main configuration is loaded from a file, and certain config values or whole sections can be overridden via command line arguments.
11. **Configurable Python references and singletons.** It should be possible to refer to Python objects (functions, classes, constants, objects) and create global singletons from callable references (functions, classes).

²Configuration files are (possibly nested) key-value pairs. We call the names of configurable hyperparameters as "config keys".

³CLI - Command Line Interface

2. Related Work

In this section we provide a brief overview of existing configuration systems. We first describe what typical ad hoc solutions look like, and then look at three config systems which are commonly used⁴ for ML workloads: OmegaConf [2], gin-config [1], and Hydra [3]. gin-config is explicitly designed for ML work, while OmegaConf and Hydra are generic config systems.

2.1. Ad Hoc Systems

Machine learning projects that do not use a specialised config system tend to consist of Python scripts where hyperparameters are defined as command line arguments. Libraries like `argparse`⁵ and `click`⁶ can be used for easier parsing of command line arguments or providing arguments via environment variables. The individual config values get passed to downstream functions where necessary.

Alternatively, a regular Python dictionary containing the configuration might be read from a YAML/JSON file, and passed along to functions that depend on it. Either the full configuration object or individual config values might be passed along, depending on programming style. It is generally not easy to unify such CLI-based and file-based configuration styles without using a specialised config system.

2.2. OmegaConf

OmegaConf builds on YAML file format, and adds a powerful interpolation mechanism, which enables accessing config keys from other parts of the file. The below example shows both absolute (`server.host`) and relative (`server.url`) references to other parts of the config file.

```
server:
  host: localhost
  port: 80
client:
  url: http://${server.host}:${server.port}/
  description: Client of ${server.url}
```

OmegaConf configuration is represented as a Python object, which can be accessed as a nested object or dictionary:

⁴By "commonly used" we mean systems we were able to find by doing relevant Google searches, reading relevant discussion threads, and looking at GitHub activity for these projects.

⁵<https://docs.python.org/3/library/argparse.html>

⁶<https://click.palletsprojects.com/>

```

conf = OmegaConf.load('conf.yaml')
assert conf.server == "localhost"
assert conf["client"]["description"] == \
    | "Client of http://localhost:80/"

```

2.3. gin-config

gin-config makes use of `@gin.configurable` decorators around function and class definitions. For functions (and initialisers of classes) decorated like this, arguments are substituted from the global config which is initialised with `gin.parse_config_file("conf.gin")`. Attributes with default value of `gin.CONFIGURED` need to have a config specified in the `.gin` config file; other arguments can (but do not have to be) configured with `gin`.

```

import gin
import torch

```

```

@gin.configurable
def dnn(
    num_inputs,
    num_outputs,
    layer_sizes=[20, 10, 20],
    activation_fn=gin.CONFIGURED,
):
    return torch.nn.Sequential(...)

```

```

gin.parse_config_file('config.gin')
model = dnn(64)

```

The config file is a custom text format, which is a simplified subset of Python. Using `dnn.num_outputs = 10` would ensure all functions named `dnn` will have the value of `num_outputs` substituted as 10. Using `path.to.mymodule.dnn.num_outputs = 10` would ensure it happens only to the function in the `path.to.mymodule` module. As special syntax, values that start with "@" refer to other `gin-configurable` functions or classes; values that start with "@" and end with "()" first get called before being passed as arguments.

```

dnn.layer_sizes = (1024, 512, 128)
path.to.mymodule.dnn.num_outputs = 10
dnn.activation_fn = @tf.nn.tanh

```

2.4. Hydra

Hydra is a feature-rich config system. The entrypoint function of the program using Hydra should be annotated with a `@hydra.main` decorator, which defines the directory where config files are stored, and config name (filename without the `.yaml` extension in the directory). When called, the function receives an `OmegaConf` `cfg`

object as an argument, which can be passed to downstream functions. Therefore the config files support all the syntax and functionality of `OmegaConf`, and some features that Hydra adds. For detailed examples, refer to Hydra documentation⁷.

```

import hydra
from omegaconf import DictConfig, OmegaConf

@hydra.main(
    version_base=None,
    config_path="conf",
    config_name="config",
)
def my_app(cfg : DictConfig) -> None:
    print(OmegaConf.to_yaml(cfg))

if __name__ == "__main__":
    my_app()

```

3. Overview of confr

In this section, we show the basics of how to use `confr`. For complete and up-to-date documentation see <https://github.com/mattiasarro/confr>.

3.1. Basic Usage

In `confr`, configs are initialised similarly to `gin-config`: functions and classes can be decorated with `confr.bind`, which ensures that function and class initialiser arguments will be substituted from the currently active config. The following example shows a function that expects at least the `num_outputs` key (and optionally `layer_sizes`) to be defined in `conf/base.yaml`. Before calling any `confr`-configured functions, `confr.init` must be called, which creates an implicit global config object.

```

import confr
import torch

@confr.bind
def dnn(
    num_inputs, # not substituted by confr
    num_outputs=confr.value,
    layer_sizes=confr.value(default=[20, 10, 20]),
):
    return torch.nn.Sequential(...)

# same as confr.init(), reads conf/base.yaml
confr.init(conf_dir="conf", base_conf="base")
model = dnn(64)

```

⁷<https://hydra.cc/docs/intro/>

All config files use the special form of YAML used by OmegaConf (with a few special cases described in the next subsections). In the default case, the arguments get bound to top-level config keys of the same name in the YAML file. So in the above example, our `conf/base.yaml` could look like this:

```
num_outputs: 5
layer_sizes: [20, 15, 10, 15, 20]
```

3.2. Custom Config Key to Argument Mapping

In some cases, a one-to-one mapping of config keys to function argument names can be limiting, so we offer two ways to customise this. Assume we have the following config:

```
num_chars: 10
neural_net:
  num_outputs: ${num_chars}
  layer_sizes: [20, 15, 10, 15, 20]
```

When wrapping a function with `confr.bind`, we could tell it to only map keys under the `neural_net` config key:

```
@confr.bind(subkeys="neural_net")
def dnn(
    num_outputs=confr.value,
    layer_sizes=confr.value(default=[20, 10, 20]),
):
```

The other option is to pass the full path in the config to `confr.value`:

```
@confr.bind
def dnn(
    num_outputs=confr.value("num_chars"),
    layer_sizes=confr.value("neural_net.layer_sizes"),
):
```

3.3. Python References and Singletons

`confr` adds special syntax to the YAML format supported by OmegaConf, which can be used to load Python object or initialise global singletons. Config values which start with a "@" are "Python references"⁸. A common use

⁸References to Python objects and references to other config values in the file look quite similar, since they both use dot notation. Python references start with a "@", and they refer Python modules like in absolute imports (which usually correspond to folder structure). Referencing other parts of a config file start with a "\$" and refer to the "path" in the YAML file, which follows the nesting of config keys.

for this is to make a preprocessing or augmentation function as a configurable argument, so that you can try out different preprocessors without changing the code. For example:

```
# config file
preprocessing_fn: @my.module.resize_and_crop

# Python code
@confr.bind
def predict(
    model,
    img,
    preprocessing_fn=confr.value,
):
    img = preprocessing_fn(img)
    return model(img)
```

When `predict` is called, the Python module `my.module` is imported and the `resize_and_crop` attribute is read from it. Any Python object could be referenced in config files - function, class, variable, constant - as long as its module is available on `PYTHONPATH`, which usually includes all modules in the project root as well as installed libraries such as `tensorflow`.

Config values which start with a "@" and end with "()" are **singletons** - Python references which get called before becoming part of the current active config and being passed as keyword arguments. For example you might define an encoder: `@my.module.my_encoder()` key-value pair in the config. Now if a function defines an argument as `encoder=confr.value`, then `@my.module` gets imported and its `my_encoder()` callable gets called before being passed as the argument value. Once `my_encoder()` is called, its return value gets memoized and any subsequent functions which use the encoder config will receive the same, pre-initialised object. Singletons can be referenced in other parts of the config using the familiar OmegaConf format of `$(config_key.subkey.singleton)$`.

Note that `my_model1` and `my_model2` in the following listing are the same object.

```
@confr.bind
def get_model1(encoder=confr.value):
    return encoder

@confr.bind
def get_model2(model=confr.value("encoder")):
    return model

my_model1 = get_my_model1()
my_model2 = get_my_model2()
assert my_model1 == my_model2
```

3.4. Scoped Arguments in Singletons

If you would like to configure input arguments specifically for singletons, you can do the following:

```
my_model1: "@my.module.model1()"
my_model1/location: "/path/to/weights.h5"
my_model2: "@my.module.model2()"
my_model2/location: "/path/to/weights2.h5"
```

Now `my_model1` singleton will be initialized with `location="/path/to/weights.h5"` and `my_model2` singleton will be initialized with `location="/path/to/weights2.h5"`. This way they can both define an input argument called `location` and still receive a unique value at initialization time. We call `my_model1/location` as a scoped argument, i.e. the value of `location` is present in only the `my_model1` singleton scope.

Note that you can still use the regular, non-scoped arguments along with scoped ones. For example, both `my_model1` and `my_model2` might define `img_h=confr.value`, and this value will be the same when initializing both singletons.

3.5. Call-time Overrides

When running a Python program configured with `confr`, individual values can be overridden in two ways:

1. Passing command-line arguments such as `--key1.subkey1=value`.
2. Setting environment variables such as `confr__key1__subkey1=value`.

3.6. Run-time overrides

When working in a notebook, modifying the YAML file to change the active config is cumbersome. You could instead initialize the config at the start of the notebook by selectively providing overrides to the keys you care about like this:

```
confr.init(
    base_conf="base",
    overrides={"override_key1": "v1"},
)
```

If you do not want to re-initialise the whole config, but would like to set individual config values, use `confr.set("my_key", value)`. Doing this would not re-initialise other config keys or singletons that may depend on `my_key`.

You may also want to provide overrides to config values temporarily, for the duration of calling a function (and any downstream functions called by this function). For example, you might want to iterate over a

list of `p_thresh` values and calculate accuracy for each `p_thresh`.

Our first attempt at solving this would look like this:

```
def precision(
    pred,
    lab,
    p_thresh=confr.value,
):
    ...

for p_thresh in p_thresholds:
    p = precision(pred, lab, p_thresh=p_thresh)
```

This would work if `precision` is the only place that uses the `p_thresh` that is passed in. But if `precision` calls `sub_function` whose `p_thresh` value comes from `confr`, then the value of `p_thresh` in `sub_function` will be the same as in the config file and not the one we passed to `precision`. What we need here is to temporarily set the value of `p_thresh` config key in the whole `confr`, like this:

```
for p_thresh in p_thresholds:
    with confr.modified_conf(
        p_thresh=p_thresh
    ):
        p = precision(pred, lab)
```

3.7. Config Spanning Multiple Files

Suppose you have the following config files:

```
# conf/base.yaml
conf_key: 123
neural_net:
    _file: shallow

# conf/neural_net/shallow.yaml
num_outputs: 10
layer_sizes: [20]

# conf/neural_net/deep.yaml
num_outputs: 10
layer_sizes: [20, 15, 10, 15, 20]
```

Once the config is loaded, the effective final config would look like this, because the `_file` special key tells `confr` to take the configuration for `neural_net` subkeys from another file.

But you could also override which neural net config gets used, by passing `--neural_net._file=deep` when running the program. Note that there is a convention between the config keys and the folders from where `_file` references are searched.

```
# conf/base.yaml
conf_key: 123
neural_net:
  num_outputs: 10
  layer_sizes: [20]
```

3.8. Accessing the Active Config

Sometimes we need to explicitly fetch the value of a key in our config system. You can use `confr.get` and `confr.set` accessors to modify the current active config:

```
import confr

confr.init(conf={"key1": "val1"})

confr.get("key1") # returns "val1"
confr.set("key1", "overwritten")
confr.get("key1") # returns "overwritten"

# can also write novel keys
confr.set("key2", "val2")
confr.get("key2") # returns "val2"
```

You can also save the current active config as a YAML file, for example at the end of training. The code for training the model and doing inference should be in the same version control project; train-time and inference-time pre-processing should be handled by the same function(s). This way, if code for inference is initialised from the same code revision and active config that was used during training, there would be no accidental difference between train time and test time behaviour.

```
confr.write_conf_file("my_active_conf.yaml")
```

3.9. Validation

Two types of validations can be done with `confr`. Each config file with name `filename.yaml` can have an optional `filename_types.yaml` counterpart, which defines the datatype of all (or a subset of) the config keys. Currently, only primitive Python types are supported, but more complex solutions will be added.

For example:

```
# conf.yaml
num_chars: 10
neural_net:
  num_outputs: ${num_chars}
  layer_sizes: [20, 15, 10, 15, 20]

# conf_types.yaml
num_chars: int
neural_net:
  layer_sizes: list
```

However, once the config gets complex enough, there is a need to validate different combinations of config values. For example, imagine we have the following config, which states that 50% of the samples come from labelled dataset, 25% come from data generator 1 and 25% come from generator 2:

```
batch_size: 32
samples_per_batch:
  labelled: 16
  gen:
    generator1: 8
    generator2: 8
```

In `confr` we can define a validator that ensures that everything in `samples_per_batch` sums to `batch_size`.

```
# my/module/main.py
from my.module import conf_validation
confr.init(validate=conf_validation)

# my/module/conf_validation.py
import confr

@confr.bind
def validate_batch_size(
  batch_size=confr.value,
  samples_per_batch=confr.value,
):
  assert batch_size == (
    samples_per_batch["labelled"] +
    sum(samples_per_batch["gen"].values())
  )
```

4. Evaluation

We will now evaluate the competing systems and confr on the desired qualities outlined in the introduction, giving each a somewhat subjective **BAD** / **OK** / **GOOD** mark.

1. Minimise boilerplate code.

- **BAD**: ad hoc, OmegaConf. Passing down configuration dictionaries or individual config values can be very verbose. So can be setting up CLI arguments in ad hoc systems.
- **OK**: Hydra. Different functions can request a config object, and read individual keys from it. However generally there is a single config object that gets passed along.
- **GOOD**: gin, confr. Configurable function arguments receive a value directly from the config system, which is most concise.

2. Minimise repetitive config.

- **BAD**: ad hoc. No way to reuse / refer to other values in YAML/JSON files.
- **OK**: gin-config. It is possible to reuse values, but in a cumbersome way, and the config file format is somewhat verbose (since all occurrences of a config value need to be listed).
- **GOOD**: OmegaConf, Hydra, confr. It is possible to reuse values and create concise config files.

3. Composability of config objects.

- **BAD**: ad hoc, OmegaConf, gin-config. Not supported.
- **GOOD**: Hydra, confr. Supported.

4. Low maintenance overhead.

- **BAD**: ad hoc, gin-config. Passing along config objects/values slows down refactoring. gin-config also often requires config changes when the relevant code changes (renaming/moving functions). Ad hoc systems might need to maintain CLI argument lists.
- **OK**: OmegaConf, Hydra. Moving code around requires changes to passing of config values, but this is less troublesome than changes required by gin-config or maintaining CLI argument lists in ad hoc systems.
- **GOOD**: confr. Config files and other parts of the source generally do not need to be changed on renaming or moving functions, because the config file makes no assumptions about where the config is used and config values do not need to be propagated.

5. Low cognitive load.

- **BAD**: gin-config, Hydra. It is not clear, without reading the configuration file and upstream code carefully, which arguments are configured and where the value comes from.
- **OK**: ad hoc, OmegaConf, confr. The initial cognitive load of understanding how a configuration is loaded in ad hoc systems and OmegaConf is low (for example, it's easy to understand what reading a YAML file or using argparse does). However it is much harder to reason about how the whole system behaves due to the next three qualities. In confr, it takes more effort to think about many possible places where configurations can come from (multiple files, command line overrides), but it is easier to reason about the whole system due to the following three qualities.

6. Clearly identifiable configurable arguments.

- **BAD**: ad hoc, OmegaConf, Hydra. When reading code (that is "far away" from the part that initialises config), it is not clear which arguments are configurable.
- **OK**: gin-config. It is possible to make it explicit that some arguments should receive a value from configuration, but this is not a requirement.
- **GOOD**: confr. It is intentionally not possible to configure an argument in confr without making it explicit in code that the value is configurable.

7. Consistent mapping between config keys and variables / arguments.

- **BAD**: ad hoc, OmegaConf, gin-config, Hydra. In all these systems such consistency is not encouraged, which makes it harder to read, understand and refactor.
- **GOOD**: confr. Such consistency is enforced by default, though in rare cases it is possible to bypass this (for example when you need to use an externally-provided config file).

8. Global config values.

- **BAD**: ad hoc, OmegaConf, Hydra. It is easy to have multiple configuration objects, or to have a different value in different places for the same config key.
- **GOOD**: gin-config, confr. There can be only one globally active configuration. In confr, each config key always has the same value.

9. **Centralised, multi-key validation of config.**

- **BAD:** gin-config. No validation system provided, hard to add one.
- **OK:** ad hoc, OmegaConf, Hydra. Centralised validation system can be added through user code to OmegaConf or file-based ad hoc systems, though this is harder to do in CLI-based ad hoc systems in a concise way. Hydra has a validation system, but this is mostly limited to data type based checks, which are insufficient.
- **GOOD:** confr. Supported, built in.

10. **Usable in a notebook as well as CLI.**

- **BAD:** ad hoc, OmegaConf, gin-config. Ad hoc systems tend to work well either for notebook-based approaches (e.g. reading a global YAML file as a config dict), or work as a sophisticated CLI script, but not both. There is no support to override options from the CLI in OmegaConf and gin-config.
- **GOOD:** Hydra, confr. Can be used in both settings.

11. **Configurable Python references and singletons.**

- **BAD:** ad hoc. Usually not supported.
- **OK:** OmegaConf, Hydra. Supported, but cumbersome to use.
- **GOOD:** gin-config, confr. Supported, easy to use.

There are features not supported by confr which other libraries provide that will be added in later versions: tab completion and more detailed type checking provided by Hydra. Hydra also has some features that were intentionally not made part of confr, such as multi-run options, custom working dir and logger config, since these were not considered relevant with respect to the evaluation criteria, which we considered most useful for a ML-oriented config system.

References

[1] Holtmann-Rice, D., Guadarrama, S., Silberman, N.: Gin Config. <https://github.com/google/gin-config> (2018)

[2] OmegaConf. <https://github.com/omry/omegaconf> (2012)

[3] Yadan, E.: Hydra - A framework for elegantly configuring complex applications, <https://github.com/facebookresearch/hydra> (2019)

The results are summarised in the following table. From it we can see that confr exhibits the good qualities of Hydra and gin-config, while alleviating some of the downsides one or the other. This is not a coincidence, since confr was in many ways inspired by these two, though the exact way confr achieves these qualities may be different.

	ad hoc	Omega	gin	Hydra	confr
1 Min boilerplate	Red	Red	Green	Orange	Green
2 Min rep conf	Red	Green	Orange	Green	Green
3 Composability	Red	Red	Red	Green	Green
4 Maintenance	Red	Orange	Green	Green	Green
5 Cognitive load	Orange	Orange	Red	Red	Orange
6 Clear args	Red	Red	Orange	Red	Green
7 Consistent keys	Red	Red	Red	Red	Green
8 Global config	Red	Red	Green	Red	Green
9 Validation	Orange	Orange	Red	Orange	Green
10 notebook + CLI	Red	Red	Red	Green	Green
11 Python ref	Red	Orange	Green	Orange	Green